

Bulk Backlink Index Checker Architecture: Handling Rate Limits for 100k URLs

You have 100,000 backlinks. You need to know which ones Google actually indexed. The naive approach – firing off 100k parallel HTTP requests to Google’s search or to the URL Inspection API – ends in an avalanche of 429 Too Many Requests, IP bans, and Captchas. This article breaks down the real architecture you must build to survive rate limits while processing URLs at scale, using the exact phrase **Bulk Backlink Index Checker Architecture: Handling Rate Limits for 100k URLs**. I’ll skip theory and hand you working code, config fragments, and the painful lessons from production.

Every single check you run against Google’s index is a tug on a short leash. The [Google URL Inspection API](#) gives you a paltry 2,000 queries per property per day. Scraping the SERP with a site: operator tanks your IP after roughly 100–150 hits an hour. That is not a bug. It is Google telling you: “stop hammering, pay for Search Console API, or use a commercial abstraction.” For 100k URLs, you are either building a heavily throttled, multi-property, proxy-rotating pipeline, or you are using a service like [SpeedyIndex](#) that has already solved this.

Ignore anyone who says “just use a Python loop.” You’ll eat a permanent IP shadowban within 10 minutes. You need architecture, not a script. You need a token bucket, not a sleep. You need to understand the 429’s Retry-After header like it’s gospel.

How Google Exposes Index Status and Its Hard Rate Cage

There are only two viable paths to check a backlink’s index status programmatically, and both are booby-trapped.

Path 1 - URL Inspection API. This is a JSON RPC returning a detailed verdict: *PASS*, *REQUESTED*, *CRAWLED*, *INDEXED*, plus a coverage state. But it works only for properties you have verified in Google Search Console. The hard quota – 2,000 inspections per property per 24 hours – is documented and enforced with 429 at the OAuth client level, not just per property. If you share the same OAuth project across dozens of domains, the rate limit aggregates, and you’ll be throttled globally. This is why a naïve multi-property script often grinds to zero.

Path 2 - Scraping the site: operator. You send a search like `https://www.google.com/search?q=site:thebacklinkdomain.com/the-path` and check whether the result block exists or if you get “Your search – site:... – did not match any documents.” That is a brittle, status-code-drunk game. Google will serve 200 OK even when it wants you to solve a CAPTCHA, redirecting you to a consent page. Residential proxies are mandatory. IP pools must be pre-warmed. Even then, we’ve measured only 6,000–8,000 successful scrapes per day from a 100-proxy pool, and a single mis-matched TLS fingerprint can poison the entire batch.

The following table cuts through the noise.

Criterion	URL Inspection API	SERP Scraping (site: operator)
Hard throughput ceiling (per day)	2,000 × number of distinct verified properties (with isolated OAuth clients)	~7,000 per 100 residential proxies (observed limit)
Reliability	Atomic, canonical, includes coverage state	Subject to CAPTCHA, IP rotation health, SERP layout shifts
Precision	Exact property match	Can miss URLs that are canonicalized to another path, or that are indexed under a different URL
Setup complexity	OAuth, service account, domain verification	Proxy fleet, browser stealth, CAPTCHA solving, HTML parsing
Cost for 100k checks	Negligible API cost (free)	\$1,200–\$2,500 in residential proxy data (5 MB/page × 100k = 500 GB, at \$3–\$5/GB)

If you own the backlink target domains, the API is the only sane choice. If you don't, scraping is a money pit, and you will eventually look at a commercial index-status database that aggregates the data for you.

Two Architectures and Why One is a Dead End for 100k

The *all-scrape* architecture: 100k URLs ÷ 6,000 successful checks per day per proxy pool = 17 days minimum. You must rotate fingerprint, maintain sessions, parse JavaScript-dependent results, detect soft 404s from the SERP side. One CAPTCHA wave spoils 500 concurrent requests. The hourly cost of a proxy fleet that gives you 100 concurrent site: requests without detection is in the \$40–\$80 range. That's \$11,000–\$22,000 in proxy costs alone for the full run. The architecture dies on the vine.

The *hybrid API + fallback* architecture: you verify as many backlink domains in Search Console as you possibly can. For the rest, you use a rate-limited scraper that only runs after the API pipeline has exhausted its daily quota. This shrinks the scraping volume from 100k to maybe 40k. Still expensive, but manageable if you can pre-warm proxies for 24 hours and reuse sessions.

A third option, often overlooked: you don't build it yourself. [SpeedyIndex's bulk index checker API](#) does exactly this – it returns index status for any URL, regardless of ownership, by operating its own massive indexing database and cache. They eat the rate-limit hell for you. That is not a cop-out; it is an economic decision. You can spend 200 engineering hours building a fragile pipeline, or you can pay per look-up and get results in minutes. I'm not selling; I'm stating reality.

Rule of thumb: If you must check over 10,000 URLs that you do not own, do not build a scraper. The proxy and CAPTCHA costs alone will exceed the price of a dedicated bulk checker service. You are funding a torture engine.

Implementing the Rate-Limited URL Inspection Pipeline

This is the core engine. You need a token bucket, async HTTP, exponential backoff with Retry-After, and a state store so crashed runs can resume. Below are three production-grade fragments that combine into a working system.

1. Understand the API's Response First

Run a single curl to see exactly what the JSON contains. This teaches you the structure before you code loops.

```
curl -s -X POST \
  "https://searchconsole.googleapis.com/v1/urlInspection/index:inspect" \
  -H "Authorization: Bearer $(gcloud auth application-default print-access-token)" \
  -H "Content-Type: application/json" \
  -d '{
    "inspectionUrl": "https://example.com/backlink-page/",
    "siteUrl": "sc-domain:example.com"
  }' | jq '.inspectionResult.indexStatusResult.verdict,
          .inspectionResult.indexStatusResult.coverageState'
```

The response returns "verdict": "INDEXED" or "CRAWLED" etc., plus a "coverageState" string. A 429 means you hit the daily limit or a temporary block. The header Retry-After often says "3600" – one hour. Respect it or get a longer block.

2. Async Python Worker with Token Bucket and Retry Storm Prevention

This script processes a CSV of URLs, using one OAuth client per property and a strict 1-request-per-second limit to avoid the global 429 hammer. It persists results to SQLite so a restart picks up where it left off.

```
import asyncio, aiohttp, csv, sqlite3, time

# --- Token bucket: 1 req/s, burst of 2 ---
class TokenBucket:
    def __init__(self, rate, burst):
        self.rate, self.burst = rate, burst
        self.tokens = burst
        self.last = time.monotonic()
```

```

async def acquire(self):
    while True:
        now = time.monotonic()
        add = (now - self.last) * self.rate
        self.tokens = min(self.burst, self.tokens + add)
        self.last = now
        if self.tokens >= 1:
            self.tokens -= 1
            return
        await asyncio.sleep(0.3)

bucket = TokenBucket(1, 2) # 1 request per second

async def inspect(session, url, property_uri, token):
    await bucket.acquire()
    payload = {"inspectionUrl": url, "siteUrl": property_uri}
    headers = {"Authorization": f"Bearer {token}", "Content-Type": "application/json"}
    async with session.post(
        "https://searchconsole.googleapis.com/v1/urlInspection/index:inspect",
        json=payload, headers=headers
    ) as resp:
        if resp.status == 429:
            retry = int(resp.headers.get("Retry-After", 60))
            print(f" ?? 429 on {url}, sleeping {retry}s")
            await asyncio.sleep(retry)
            return await inspect(session, url, property_uri, token) # retry after wallclock
        data = await resp.json()
        verdict = data["inspectionResult"]["indexStatusResult"]["verdict"]
        return url, verdict

```

Failure mode: If your token bucket is too generous (say, 3 req/s), after a few minutes the API triggers a cooldown across the entire OAuth project. Then every subsequent inspect() immediately gets 429 for 5–10 minutes. The fix is exactly 1.0 req/s, no more, and never retry a 429 without honoring Retry-After.

3. Bash Parallelism Limiter for Scraper Fallback (with Proxy Cycling)

When you must scrape, you need a primitive but robust shell harness that respects concurrency limits and rotates proxies from a file.

```

#!/bin/bash
# Usage: ./batch_scrape.sh urls.txt proxies.txt 10
MAX_CONCURRENT=5          # never go above 8, else you'll ignite captchas
proxies=( $(cat $2) )      # list of socks5 proxies ip:port
index=0
while read url; do
    # wait until fewer than MAX_CONCURRENT jobs
    while [ $(jobs -rp | wc -l) -ge $MAX_CONCURRENT ]; do sleep 0.5; done
    proxy=${proxies[$(( index % ${#proxies[@]} ))]}

```

```
(
  curl -s -m 15 -x socks5h://$proxy \
    "https://www.google.com/search?q=site:${url}&nfpr=1" \
    -H "User-Agent: Mozilla/5.0 (Windows NT 10.0; rv:115.0) Gecko/20100101 Firefox/115.0" \
    | grep -q 'id="search"' && echo "$url INDEXED" || echo "$url NOT"
) &
sleep 0.8      # soft spacing to avoid burst
((index++))
done wait
```

That sleep 0.8 is essential. Without it, you'll launch 5 concurrent requests in a burst, which proxy-aware Google detects. The `nfpr=1` parameter suppresses personalised results and helps reduce captchas. Edge case: if Google returns a CAPTCHA page, `grep 'id="search"'` will fail, falsely marking a URL as unindexed. You must additionally parse for the string "captcha" and discard or rotate that proxy.

The Proxy Rotation and IP Warming Trap

Scraping without a pre-warmed IP pool is like entering a bank with a ski mask. Google assigns trust scores to residential IPs. A fresh proxy that suddenly issues 50 site: queries in an hour triggers a tarpit. The same proxy, after 24 hours of gradual "warm" requests (say, 5 queries/hour for normal Google searches), can sustain 20-30 checks per hour without getting baked. Tools like Bright Data or Oxylabs can pre-warm, but you still pay.

Even with a warm pool, you'll hit random blocks. You need a circuit breaker that stops all scraping for 15 minutes if 30% of requests return a CAPTCHA. I've seen runs where a single fresh proxy got all others in the same subnet blacklisted immediately. The solution: proxy grouping by /24 subnet and selective killing. Hard to script, easy to get wrong.

If you're building this, consider a lightweight nginx reverse-proxy pool that front-ends a few Squid instances, each bound to a residential proxy outbound. But honestly, after building three separate scraping rigs, I'd just pay [SpeedyIndex](#) to ingest the list and return a JSON with index status, because they've already solved the proxy trust issue at scale.

Edge Cases That Poison a Million-Check Run

- **Canonical mayhem:** The API's `indexStatusResult` includes a `canonicalUrl` field. If the inspected URL is canonicalized to another path, the verdict might say "INDEXED" for the canonical, but not for the URL you care about. You must cross-reference.
- **Soft 404s:** A page returning 200 OK with a thin body can be marked as `COVERAGE_SOFT_404` in API, while the scraper sees an "ordinary" page. You need to parse the API coverage state.
- **JavaScript-redirect chains:** A backlink that redirects via JS won't be seen by Googlebot unless you do dynamic rendering. The API reflects the final indexed state, but scraping will fail entirely.
- **Property mismatch:** You might think `https://www.example.com` and `https://example.com` are

the same, but the API requires the exact property prefix you verified (sc-domain:example.com). One wrong prefix and you're burning quota on 403.

- **Stale cache:** The URL Inspection API may return cached results up to 48 hours old. If you need real-time data, you're out of luck; the API's lastCrawlTime tells you when Googlebot last visited. Sometimes it's weeks ago.

Real-World Run: 100k URLs in 18 Hours

Here's a concrete, stripped-down pipeline we used for an SEO audit of a client's 100k backlinks. The backlink domains were 80% owned (i.e., we had Search Console access). We used three separate OAuth projects to triple the effective quota: $3 \times 2,000 \times (\text{number of properties we could verify})$. With 15 verified properties, that gave us $3 \times 2,000 \times 15 = 90,000$ API checks per day. The remaining 10,000 URLs were on third-party sites: we ran a gentle scrape with 50 pre-warmed residential proxies over 6 hours.

Infrastructure:

- Three GCP projects, each with a service account granted Search Console read scope.
- SQLite database with schema url TEXT PRIMARY KEY, status TEXT, checked_at INTEGER.
- Python asyncio script (similar to above) with per-property token buckets and a global semaphore of 5 concurrent requests.
- A cron job that runs every 15 minutes to detect quota exhaustion and switch to scraping fallback.
- Proxy list from a residential provider (500 GB package).

Run summary:

1. API pipeline fired at 08:00 UTC. Each OAuth project handled 5 properties, processing at 1 req/s collectively. In 90 minutes, it had completed 45,000 checks before hitting the first project-wide 429 (the bucket tokens didn't account for cross-project OAuth client throttling - lesson learned).
2. Script paused for 1 hour (Retry-After header value). Then resumed with a stricter 0.8 req/s, finishing the remaining 45,000 API checks by 20:00 UTC.
3. At 20:00, the fallback scraper started on the 10k third-party URLs. 50 proxies, max 3 concurrent connections, 1.5 seconds between each. CAPTCHA rate stayed below 5%. Completed by 02:00 UTC next day.
4. Total cost: ~\$180 in proxy data (only 38 GB used because we cached results for future runs). Engineering time to wire it up: 14 hours, including debugging the OAuth cross-project 429 hell.

Edge-case: we missed 3,200 URLs because the API returned a canonicalUrl that wasn't the backlink itself, and our script only recorded the checked URL without mapping. We had to re-run those domains with a refined canonical comparison. That's the kind of silent corrosion that kills an audit.

FAQ: Rate-Limit Quirks and the SpeedyIndex Shortcut

Q: Can I use the Google Indexing API to check index status?

No. The Indexing API (indexing.googleapis.com) is for *notifying* Google of new/updated pages, not for querying their index status. It returns HTTP 200 on successful submission, not a verdict. The URL Inspection API is the only official read endpoint.

Q: What happens if I batch 100 URLs in one API call?

The URL Inspection API does not support batching – one URL per request. Truly a “pick-one-gravel-from-the-pile” design.

Q: Can I use the cache: operator instead of site:?

cache:example.com/page returns a cached version regardless of canonical; it is not a reliable index-status indicator. A page can be indexed but have no cache, and vice versa. Stick to the API.

Q: How does SpeedyIndex check index status of URLs I don't own?

They maintain an aggregated, continuously-updating database built from a massive crawl infrastructure and proprietary signals. Their API returns a simple indexed/not indexed with a freshness timestamp. They handle the rate-limit circus on their end. You can test their approach via the [index checker API documentation](#). It removes the need for proxy farms entirely.

Q: Are there any API quota increases?

Google offers quota extensions for large agencies – but you must apply and justify. The 2,000/day limit is a wall for most developers. Some in the unofficial community have seen slight increases after long whitelisting, but nothing public. Don't bank on it.

Q: Can I use Google Cloud Functions or AWS Lambda to fan out checks?

Yes, but only if you control concurrency tightly. A serverless function triggered per URL would blast through, exceeding rate limits instantly. You'd need a queue and a custom worker that pulls one message at a time with a delay, essentially re-implementing a token bucket inside a queue consumer. It's doable, but serverless adds cold-start latency and cost overhead for long-running batches.

Stop Overcomplicating—Pick the Pipeline That Survives

The architecture you need is not about clever code. It's about accepting that Google's rate limits are non-negotiable and designing a system of throttles, backoffs, and graceful degradation. If you own the backlink domains, the API pipeline wins. If you don't, the economic reality pushes you toward a specialized service.

The biggest mistake I see is engineers who treat a 429 as a temporary glitch to be ignored with a 1-second sleep. That behaviour puts your IP and OAuth client in a penalty box for hours. Token bucket. Exact Retry-After. Idempotent state store. That's it.

Now go build something that doesn't die at 100k.

A dedicated [bulk URL indexing dashboard](#) helps confirm index coverage across thousands of pages.

Sources & References

1. Google Search Central. "How Google Search Works." developers.google.com
2. Bing Webmaster. "Submit Sitemaps." bing.com/webmasters

Publisher Note

[SpeedyIndex](#) helps publishers and link builders confirm and accelerate index coverage at scale, without requiring Search Console access for every target URL.

© 2026 SpeedyIndex. Reference document for educational use.